Propositional Logic

0.1 Definitions

- Implication: When P is true, Q must be true. False implies true.
- $P \implies Q \equiv \neg P \vee Q$
- If σ is an assignment and φ is a formula, then $\sigma \models \varphi$ (sigma satisfies phi) means φ is true under assignment σ.
- A formula φ is satisfiable if there exists an assignment σ such that $\sigma \models \varphi$. Otherwise, we say it's unsatisfiable.
- A formula is valid (aka a tautology) if for every assignment σ, $\sigma \models \varphi$.
- Two formulas φ and ψ are logically equivalent, denoted $\varphi \equiv \psi$, if for every assignment σ, $\sigma \models \varphi$ iff $\sigma \models \psi$.
- Formulas φ and ψ are equisatisfiable if φ is satisfiable iff ψ is satisfiable.

0.2 The Boolean Satisfiability Problem (SAT)

- Given a propositional formula φ, is φ satisfiable?
- Example: System M with input x and output y and an input-output specification $\varphi(x, y)$. If we can encode the computation of M as a formula $\psi_m(x, y)$ such that this formula holds iff y is the output of M on input x (ψ_m is called the input-output relation or transition relation of M), then we can decide whether M satisfies φ for all inputs using this formula:

  $$\chi = \psi_m(x, y) \wedge \neg\varphi(x, y)$$

  The formula is satisfiable iff $M \not\models \varphi$, ie there is some input x such that M violates φ on that input. So to verify M, we can check if χ is unsatisfiable.
- Usually when we refer to SAT, we are referring to the CNF-SAT problem, where the input formula is in conjunctive normal form (CNF).
- A formula is in CNF if it is a conjunction of clauses, where each clause is a disjunction (OR) of literals, and a literal is either a variable or the negation of a variable. Example:

  $$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$
- The top level operator must be an AND.
- Any formula can be converted to CNF.
- A useful transformation is the Tseitin transformation, which converts any formula φ to an equisatisfiable CNF formula ψ in linear time. The idea is to introduce a new variable for each subformula of φ and add clauses that enforce the equivalence between the new variable and the subformula.

Example:

$$\varphi = \neg(x \vee y) \vee (\neg z \wedge x)$$

List the cases for the clause (gate) (call the output the clause u):

$$x \implies \neg u$$

$$y \implies \neg u$$

$$\neg x \wedge \neg y \implies u$$

Using demorgans law, we can rewrite these as:

$$\neg x \vee \neg u$$

$$\neg y \vee \neg u$$

$$x \vee y \vee u$$

AND them all together to represent the first gate!

$$(\neg x \vee \neg u) \wedge (\neg y \vee \neg u) \wedge (x \vee y \vee u)$$

These three clauses uniquely determine u given values for x and y (among satisfying assignments). Repeating this for every gate in the circuit gives a set of clauses encoding how the whole circuit works. Then add one more clause asserting that the output wire is true. Now any satisfying assignment of the original formula can be extended to one satisfying the new formula, and conversely any satisfying assignment of the new formula satisfies the original.

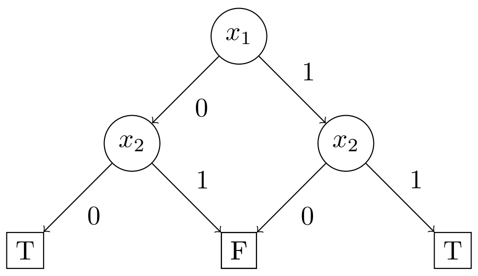
- Since there is 1 gate per boolean operator, and a constant number of clauses per gate, the new CNF formula has size linear in the size of the original formula.
- Disjunctive Normal Form (DNF) is the dual of CNF, where the top level operator is OR and each cube is an AND of literals. Example:

$$(x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_4) \vee (x_2 \wedge x_3 \wedge \neg x_4)$$

- DNF is always satisfiable unless every cube is contradictory (eg $x_1 \wedge \neg x_1$).
- SAT is solvable in linear time for DNF.
- There is no efficient way to convert an arbitrary formula to DNF.

0.3 Binary Decision Diagrams (BDDs)

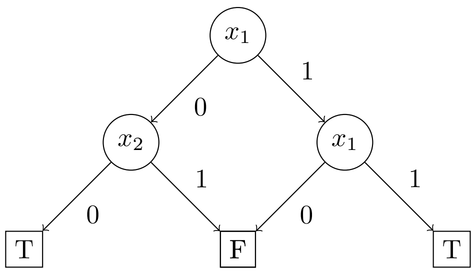
- It's a representation of boolean functions as graphs that allows varius operations (in particular, satisfiability) to be done efficiently but is more compact than a truth table.
- A BDD is a directed acyclic graph (DAG) with:
  - A designated root node
  - Leaf nodes will be labeled true or false
  - Internal nodes will be labeled with a variable and have two outgoing edges, one for the case where the variable is true and one for false.
- The BDD for iff looks like:



- Given a variable assignment, follow the corresponding edges until you reach a leaf. The value of the function is the label of the leaf.
- Every boolean function can be represented by a BDD.

Reduced Ordered BDDs (ROBDDs)

- Ordered: The variables must be accessed according to a specified order along every branch of the BDD. Skipping is okay, but you cannot repeat them. An example BDD that is not ordered:



- Reduced: No duplicate nodes/leaves or redundant nodes. A node is redundant if both its edges point to the same node.
- There is only one unique ROBDD for a given boolean function.
- You can check if two functions are equivalent by checking if their ROBDDs are identical.
- The size of the unique ROBDD for a given function can be very different depending on the variable ordering.
- How to build a ROBDD: Build a truth table, then build the fully expanded BDD, then reduce it.
- Second approach: Build from a formula in a recursive bottom-up manner by applying boolean operations. Given ROBDDs for formulas φ and ψ, we can build the ROBDD for $\varphi \wedge \psi, \varphi \vee \psi, \neg\varphi$, etc.

  To do this, we'll use Shannon Expansion (aka Boole's Law), which expands a boolean function in terms of one of its variables.

  $F(x, \cdots, x_n) \iff (x_1 \wedge F(T, x_2, \cdots, x_n)) \vee (\neg x_1 \wedge F(F, x_2, \cdots, x_n))$
- $F(T, x_2, \cdots, x_n)$ is the positive cofactor of F with respect to x_1. Written $F_{x_1}(x_2, \cdots, x_n)$.
  $F(F, x_2, \cdots, x_n)$ is the negative cofactor of F with respect to x_1. Written $F_{\bar{x_1}}(x_2, \cdots, x_n)$.
- You can get the cofactors of a BDD by following the appropriate edges.
- Every node of the BDD corresponds to the cofactor of F with respect to the assignments leading to that node.
- The cofactors of the AND of two functions are the AND of the cofactors. E.g.

$$\begin{aligned} F(x, y) &= H(x, y) \wedge G(x, y) \\ F_x(y) &= F(T, y) = H(T, y) \wedge G(T, y) = H_x(y) \wedge G_x(y) \\ F_{\bar{x}}(y) &= F(F, y) = H(F, y) \wedge G(F, y) = H_{\bar{x}}(y) \wedge G_{\bar{x}}(y) \end{aligned}$$

  It's the same for OR, NOT, etc.
- Extisential quantification: $\exists x_1.\varphi(x_1, \cdots, x_n) \iff \varphi(T, x_2, \cdots, x_n) \vee \varphi(F, x_2, \cdots, x_n)$
- Universal quantification: $\forall x_1.\varphi(x_1, \cdots, x_n) \iff \varphi(T, x_2, \cdots, x_n) \wedge \varphi(F, x_2, \cdots, x_n)$
- There are boolean functions with n variables whose ROBDDs (even with the best variable ordering) have size exponential in n.
- Idea: Counting argument. There are 2^{2^n} distinct boolean functions on n variables, but fewer BDDs with less than exponential nodes.

Queries on BDDs

- Does a given assignment/input satisfy hte formula? (Linear, simply plug into the BDD).
- You can check satisfiability and validity in constant time. Simply check if the ROBDD is just a single true or false leaf.
- Are two formulas equivalent? (Linear, check if the ROBDDs are identical).
- How many satisfying assignments are there? Idea: think about all paths from the root to the true leaf.

SAT Solving

- Motivation: Avoid BDD blowup by trading time for memory. SAT solvers don't run out of memory, but they may search for a long time.
- SAT is NP-complete. All known algorithms for SAT take more than polynomial time in the worst case. But this does not rule out practical algorithms.
- NP-completeness doesn't imply that all SAT problems take exponential time. There are some special cases which are solvable efficiently.
  - Horn clauses: CNF-SAT. Each clause has at most one positive literal. E.g.

    $$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1) \wedge (\neg x_1 \vee x_3)$$

    This is equivalent to implications whose left hand sides are conjunctions of variables. E.g.

    $$(x_1 \wedge x_2 \implies x_3) \wedge (x_2 \wedge x_3 \implies \text{false}) \wedge (\text{true} \implies x_1) \wedge (x_1 \implies x_3)$$

  - 2SAT: CNF-SAT where each clause has at most 2 literals. E.g.

    $$(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

- For general SAT, there are heuristic algorithms that tend to work well in practice even though their runtimes are exponential in the worst case.
- We'll look at search algorithms which the space of assignment looking for a satisfying assignment.
- Simplest approach: Check all 2^n assignments by brute force. Guaranteed to take exponential time if the formula is UNSAT.
- Better way: Search through partial assignments, pruning away parts of the space that can't possibly satisfy the formula.
- First algorithm: Backtracking search. Divide and conquer. Split the space on some variable, assign it to true and false respectively (reducing the number of variables by 1). Solve those subproblems recursively. If $x = F$ doesn't work, backtrack by undoing that assignment and trying $x = T$ instead.
- DPLL algorithm: Backtracking search with better formula simplification during search.
- Unit propagation: A clause with only one literal (a unit clause) forces a variable to be true/false. Clauses can become unit after a partial assignment. You then don't split on the remaining variable, you simply assign it accordingly and continue. Then we can propagate this assignment to potentially force further assignments. (Note: $T \vee \neg y$ is not considered a unit since it is already satisfied.) If UP yields a contradiction this is a conflict: An assignment might falsify a conflicting clause.

- Pure literal elimination: If a variable appears with only one polarity (always negated or always unnegated) in the formula, it can be assigned to satisfy all those clauses and removed from the formula. E.g. in

$$(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_4 \vee x_1)$$

  x_4 is a pure literal and can be set to true and removed.

- DPLL algorithm summary:

  1. $\varphi \leftarrow$ simplify(φ). Apply UP and PLE until reaching a fixpoint.
  2. If we encountered a conflict, return UNSAT.
  3. If φ is empty (all clauses satisfied), return SAT.
  4. Pick a variable x in φ.
  5. If DPLL($\varphi \wedge x$) = SAT, return SAT.
  6. If DPLL($\varphi \wedge \neg x$) = SAT, return SAT.
  7. Return UNSAT.

  Correctness: φ is satisfiable iff either $\varphi \wedge x$ or $\varphi \wedge \neg x$ is satisfiable.

- Issues: Keeping track of different simplified formulas. Finding unit clauses efficiently (naive implementation would have to scan every clause wherever we make an assignment). Can encounter the same conflict many times, searching parts of the space that can't possibly work.

- Solution: Learn a clause after each conflict that eliminates the partial assignment which led to the conflict (the reason for the conflict). Later, this "conflict clause" (here $x_1 \vee x_1 000$) will trigger unit propagation before we would hit the same conflict again.

Conflict-driven clause learning (CDCL) algorithm.

- Notation: Assigning a value to a unassigned variable when there is no UP left to do is called a decision. All assigned variables during the search are either decision variables or implied by UP. The decision level (DL) of a variable is the depth of the search tree when it was assigned.

- We'll write $\neg x_5$@7 to mean x_5 was assigned false at decision level 7.